

# 浙江大学

## 本科实验报告

课程名称: 操作系统

姓 名:

学 院: 计算机科学与技术学院

系: 计算机科学与技术系

专 业: 计算机科学与技术

学 号:

指导教师: 李环

2024 年 12 月 20 日

# 浙江大学操作系统实验报告

实验名称： VFS & FAT32 文件系统

电子邮件地址： 手机：

实验地点： 曹西 503 实验日期： 2024 年 12 月 20 日

## 一、实验目的和要求

为用户态的 Shell 提供 read 和 write syscall 的实现  
实现 FAT32 文件系统的基本功能，并对其中的文件进行读写

## 二、实验过程

### （一）准备工程

从仓库同步以下文件：

1.	src/lab6	
2.	├─ disk.img.zip	// FAT32 磁盘镜像，需解压
3.	├─ fs	
4.	│   └─ Makefile	
5.	│   └─ fat32.c	// FAT32 文件系统实现
6.	│   └─ fs.c	// 供系统内核使用的文件系统相关函数实现
7.	│   └─ mbr.c	// MBR 初始化（无需修改）
8.	│   └─ vfs.c	// VFS 实现
9.	│   └─ virtio.c	// VirtIO 驱动（无需修改）
10.	└─ include	
11.	│   └─ fat32.h	// FAT32 相关数据结构与函数声明
12.	│   └─ fs.h	// 供系统内核使用的文件系统相关数据结构及函数声明
13.	│   └─ mbr.h	// MBR 相关数据结构与函数声明（无需关注）
14.	│   └─ vfs.h	// VFS 操作函数声明
15.	│   └─ virtio.h	// VirtIO 驱动相关数据结构与函数声明（无需关注）
16.	└─ user	// 用户态程序部分不需同学们修改，所以这里给出完整的代码

```

17.      └─ Makefile      // 未作修改
18.      └─ link.lds      // 未作修改
19.      └─ main.c        // nish 用户态程序（需要阅读）
20.      └─ printf.c      // 未作修改
21.      └─ start.S        // 未作修改
22.      └─ stddef.h      // 未作修改
23.      └─ stdint.h      // 同步了内核的 stdint.h
24.      └─ stdio.h        // 未作修改
25.      └─ string.c       // 新文件，添加了 strlen
26.      └─ string.h       // 新文件，添加了 strlen
27.      └─ syscall.h      // 更新了系统调用号
28.      └─ uapp.S         // 未作修改
29.      └─ unistd.c       // 系统调用实现（需要阅读）
30.      └─ unistd.h

```

需要在 arch/riscv/Makefile 里面添加相关编译产物来进行链接：

```

1.      ${LD} -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.
      o ../../fs/*.o ../../user/uapp.o -o ../../vmlinux

```

需要对根目录下的 Makefile 作出以下修改：

```

1.      all: clean
2.      ${MAKE} -C lib all
3.      ${MAKE} -C init all
4.      ${MAKE} -C user all
5.      ${MAKE} -C fs all
6.      ${MAKE} -C arch/riscv all
7.      @echo -e '\n'Build Finished OK
8.
9.      clean:
10.     ${MAKE} -C lib clean
11.     ${MAKE} -C init clean
12.     ${MAKE} -C user clean
13.     ${MAKE} -C fs clean
14.     ${MAKE} -C arch/riscv clean
15.     $(shell test -f vmlinux && rm vmlinux)
16.     $(shell test -f System.map && rm System.map)
17.     @echo -e '\n'Clean Finished

```

## （二）Shell: 与内核进行交互

### 1. 文件系统抽象

在 include/fs.h 中定义了文件系统的数据结构：

```

1.      struct file {      // Opened file in a thread.

```

```

2.      uint32_t opened;    // 文件是否打开
3.      uint32_t perms;     // 文件的读写权限
4.      int64_t cfo;        // 当前文件指针偏移量
5.      uint32_t fs_type;   // 文件系统类型
6.
7.      union {
8.          struct fat32_file fat32_file;    // 后续 FAT32 文件系统的文
          件需要的额外信息
9.      };
10.
11.     int64_t (*lseek) (struct file *file, int64_t offset, uint64_t
        whence); // 文件指针操作
12.     int64_t (*write) (struct file *file, const void *buf, uint64_
        t len);    // 写文件
13.     int64_t (*read)  (struct file *file, void *buf, uint64_t len);
        // 读文件
14.
15.     char path[MAX_PATH_LENGTH]; // 文件路径
16. };
17.
18. struct files_struct {
19.     struct file fd_array[MAX_FILE_NUMBER];
20. };

```

修改 proc.h, 为进程 task\_struct 结构体添加一个指向文件表的指针:

```

1.  struct task_struct {
2.      uint64_t state;
3.      uint64_t counter;
4.      uint64_t priority;
5.      uint64_t pid;
6.
7.      struct thread_struct thread;
8.      uint64_t *pgd;
9.      struct mm_struct mm;
10.     struct files_struct *files;
11. };

```

## 2. stdout/err/in 初始化

fs/fs.c 文件中, 我们定义了一个函数 file\_init, 需要在 proc.c 中的 task\_init 函数中为每个进程调用, 创建文件表并保存在 task struct 中。

在这个函数中，根据 `files_struct` 的大小分配页空间，以及为 `stdin`、`stdout`、`stderr` 赋值：

```
1.  struct files_struct *file_init() {
2.      // todo: alloc pages for files_struct, and initialize stdin,
      stdout, stderr
3.      struct files_struct *ret = (struct files_struct*)alloc_page()
      ;
4.      ret->fd_array[0].opened = 1;
5.      ret->fd_array[0].perms = FILE_READABLE;
6.      ret->fd_array[0].cfo = 0;
7.      ret->fd_array[0].lseek = NULL;
8.      ret->fd_array[0].write = NULL;
9.      ret->fd_array[0].read = stdin_read;
10.     memcpy(ret->fd_array[0].path, "stdin", 6);
11.
12.     ret->fd_array[1].opened = 1;
13.     ret->fd_array[1].perms = FILE_WRITABLE;
14.     ret->fd_array[1].cfo = 0;
15.     ret->fd_array[1].lseek = NULL;
16.     ret->fd_array[1].write = stdout_write;
17.     ret->fd_array[1].read = NULL;
18.     memcpy(ret->fd_array[1].path, "stdout", 7);
19.
20.     ret->fd_array[2].opened = 1;
21.     ret->fd_array[2].perms = FILE_WRITABLE;
22.     ret->fd_array[2].cfo = 0;
23.     ret->fd_array[2].lseek = NULL;
24.     ret->fd_array[2].write = stderr_write;
25.     ret->fd_array[2].read = NULL;
26.     memcpy(ret->fd_array[2].path, "stderr", 7);
27.     return ret;
28. }
```

### 3. 处理 `stdout/err` 的写入

在捕获到 `write` 的 `syscall` 之后，我们就可以查找对应的 `fd`，并通过对应的 `write` 函数调用来进行输出了，实现如下：

```
1.  int64_t sys_write(uint64_t fd, const char* buf, uint64_t len)
2.  {
3.      uintptr_t ret;
4.      struct file *file = &(current->files->fd_array[fd]);
5.      if (file->opened == 0) {
6.          printk("file not opened\n");
7.          return ERROR_FILE_NOT_OPEN;
```

```

8.         } else {
9.             if (file->perms & FILE_WRITABLE) {
10.                 ret = file->write(file, buf, len);
11.             }
12.         }
13.         return ret;
14.     }

```

在 `trap.c` 中添加相关的系统调用：

```

1.     if (regs->x[17] == SYS_WRITE) {
2.         ret = (uint64_t)sys_write((uint64_t)(regs->x[10]), (const cha
   r*)(regs->x[11]), (uint64_t)(regs->x[12]));
3.         regs->sepc += 4;
4.     }

```

对于 `stdout` 和 `stderr` 的输出，我们直接通过 `printk` 进行串口输出即可：

```

1.     int64_t stdout_write(struct file *file, const void *buf, uint64_t
   len) {
2.         char to_print[len + 1];
3.         for (int i = 0; i < len; i++) {
4.             to_print[i] = ((const char *)buf)[i];
5.         }
6.         to_print[len] = 0;
7.         return printk(buf);
8.     }
9.
10.    int64_t stderr_write(struct file *file, const void *buf, uint64_t
   len) {
11.        char to_print[len + 1];
12.        for (int i = 0; i < len; i++) {
13.            to_print[i] = ((const char *)buf)[i];
14.        }
15.        to_print[len] = 0;
16.        return printk(buf);
17.    }

```

#### 4. 处理 `stdin` 的读取

对于输入的读取就是对于 `fd=0` 的 `stdin` 文件进行 `read` 操作，所以需要实现 `vfs.c` 中的 `stdin_read` 函数。而对于终端的输入，我们需要通过 `sbi` 来完成，需要在 `arch/riscv/include/sbi.h` 中添加函数，并在 `sbi.c` 中进行实现：

```

1.     struct sbiret sbi_debug_console_read(uint64_t byte, uint64_t addr
    _low, uint64_t addr_high) {
2.         return sbi_ecall(0x4442434E, 0x1, byte, addr_low, addr_high,
    0, 0, 0);
3.     }

```

完成了 `stdin_read` 后，还需要捕获 63 号系统调用 `read`，来和 `write` 一样类似处理即可：

```

1.     else if (regs->x[17] == SYS_READ) {
2.         ret = (uint64_t)sys_read((uint64_t)(regs->x[10]), (const char
    *) (regs->x[11]), (uint64_t)(regs->x[12]));
3.         regs->sepc += 4;
4.     }
5.
6.     int64_t sys_read(uint64_t fd, char* buf, uint64_t len) {
7.         uint64_t ret;
8.         struct file *file = &(current->files->fd_array[fd]);
9.         if (file->opened == 0) {
10.             printk("file not opened\n");
11.             ret = ERROR_FILE_NOT_OPEN;
12.         } else {
13.             if (file->perms & FILE_READABLE) {
14.                 ret = file->read(file, buf, len);
15.             }
16.         }
17.         return ret;
18.     }

```

全部完成后，就可以在 `nish` 中使用 `echo` 命令了：

```
root@oem-virtual-machine: /home/oem/Desktop/lab6
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x00000000000001666
Boot HART MEDELEG         : 0x00000000000f0b509
...buddy_init done!
...mm_init done!
...virtio_blk_init done!
...fat32 partition #1 init done!
...task_init done!
2024 ZJU Operating System
scause = 12, sepc = 100e8, stval = 100e8
scause = 15, sepc = 10aa0, stval = 3fffffff8
hello, stdout!
hello, stderr!
scause = 13, sepc = 10c00, stval = 14000
SHELL > echo "test"
scause = 15, sepc = 102d4, stval = 13000
test
SHELL > echo ""
SHELL > echo"test"
test
SHELL > echo "test"
test
SHELL >
```

### （三）FAT32：持久存储

在本次实验中我们仅需实现 FAT32 文件系统中很小一部分功能，我们为实验中的测试做如下限制：

文件名长度小于等于 8 个字符，并且不包含后缀名和字符..

不包含目录的实现，所有文件都保存在磁盘根目录 /fat32/ 下。

不涉及磁盘上文件的创建和删除。

不涉及文件大小的修改。

#### 1. 准备工作

解压 src/lab6/disk.img.zip 得到 disk.img 并放在根目录下

在 Makefile 中添加以下代码：

1. run: all
2. @echo Launch qemu...
3. @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux  
-bios default \
4. -global virtio-mmio.force-legacy=false \



```

5.         -drive file=disk.img,if=none,format=raw,id=hd0 \
6.         -device virtio-blk-device,drive=hd0
7.
8.     debug: all
9.     @echo Launch qemu for debug...
10.    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux
    -bios default \
11.        -global virtio-mmio.force-legacy=false \
12.        -drive file=disk.img,if=none,format=raw,id=hd0 \
13.        -device virtio-blk-device,drive=hd0 -S -s

```

VirtIO 所需的驱动已经编写完成了，在 `fs/virtio.c` 中给出，为了正常使用这部分外设，还需要在 `setup_vm_final` 中添加对 VirtIO 外设的映射：

```

1.     create_mapping(swapper_pg_dir, io_to_virt(VIRTIO_START), VIRTIO_S
    TART, VIRTIO_SIZE * VIRTIO_COUNT, PTE_W | PTE_R | PTE_V);

```

在 `head.S` 中 `task_init` 结束后调用 `virtio_dev_init()` 和 `mbr_init()` 进行初始化：

```

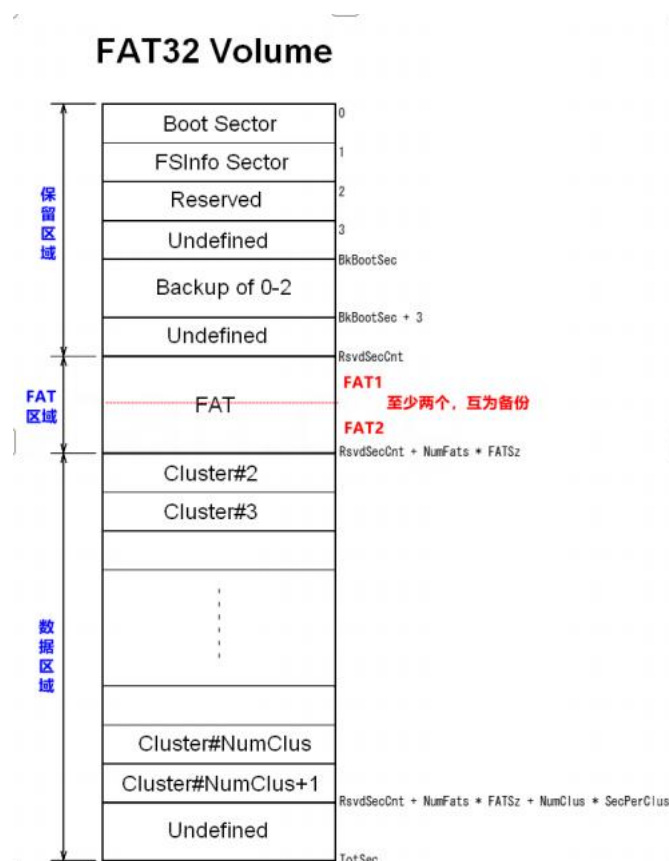
1.     virtio_dev_init();
2.     mbr_init();

```

## 2. 初始化 FAT32 分区

在 FAT32 分区的第一个扇区中存储了关于这个分区的元数据，首先需要读取并解析这些元数据。我们提供了两个数据结构的定义，`fat32_bpb` 为 FAT32 BIOS Parameter Block 的简写。这是一个物理扇区，其中对应的是这个分区的元数据。首先需要将该扇区的内容读到一个 `fat32_bpb` 数据结构中进行解析。

`fat32_volume` 是用来存储我们后续代码中需要用到的元数据的，需要根据 `fat32_bpb` 中的数据来进行计算并初始化。



计算和初始化的完成参考上图。左图显示 FAT32 的分区依次分为保留区域、FAT 区域和数据区域。观察 `fat32_bpb` 和 `fat32_volume` 各成员变量的名称，推测其含义，容易得到初始化的函数实现如下：

```

1. void fat32_init(uint64_t lba, uint64_t size) {
2.     virtio_blk_read_sector(lba, (void*)&fat32_header);
3.     fat32_volume.first_fat_sec = lba + fat32_header.rsvd_sec_cnt/
        * to calculate */;
4.     fat32_volume.sec_per_cluster = fat32_header.sec_per_clus/* to
        calculate */;
5.     fat32_volume.first_data_sec = fat32_volume.first_fat_sec + fa
        t32_header.fat_sz32 * fat32_header.num_fats/* to calculate */;
6.     fat32_volume.fat_sz = fat32_header.fat_sz32/* to calculate */
7.     ;
8. }
```

### 3. 完善系统调用

在读取文件之前，首先需要打开对应的文件，这要实现 `openat` syscall，调用号为 56。你需要寻找一个空闲的文件描述符，然后调用 `file_open` 函数来初始化这个文件描述符，同时在 `syscall.c` 中添加系统调用：

```

1.  int64_t sys_openat(int dfd, const char *filename, int flags) {
2.      int fd = -1;
3.
4.      for (int i = 0 ; i < MAX_FILE_NUMBER; i++) {
5.          if (current->files->fd_array[i].opened == 0) {
6.              fd = i;
7.              break;
8.          }
9.      }
10.
11.     file_open(&(current->files->fd_array[fd]), filename, flags);
12.
13.     return fd;
14. }

```

#### 4. 打开文件

在 `fat32_open_file` 函数中，我们需要读取被打开的文件所在的簇和目录项位置的信息，来供后面 `read write lseek` 使用，需要遍历数据区开头的根目录扇区，找到 `name` 和 `path` 末尾的 `filename` 相匹配的 `fat32_dir_entry` 目录项结构体，再从其中得到这些信息：

```

1.  struct fat32_file fat32_open_file(const char *path) {
2.      struct fat32_file file;
3.      /* todo: open the file according to path */
4.      char filename[9];
5.      memset(filename, ' ', 9);
6.      filename[8] = '\0';
7.
8.      if (strlen(path) - 7 > 8) {
9.          memcpy(filename, path + 7, 8);
10.     } else {
11.         memcpy(filename, path + 7, strlen(path) - 7);
12.     }
13.     to_upper_case(filename);
14.     uint64_t sector = fat32_volume.first_data_sec;
15.     virtio_blk_read_sector(sector, fat32_buf);
16.     struct fat32_dir_entry *entry = (struct fat32_dir_entry *)fat32_buf;
17.     for (int entry_index = 0; entry_index < fat32_volume.sec_per_cluster * FAT32_ENTRY_PER_SECTOR; ++entry_index) {
18.         char name[8];
19.         memcpy(name, entry->name, 8);
20.         for (int k = 0; k < 9; ++k) {
21.             if (name[k] <= 'z' && name[k] >= 'a') {

```

```

22.             name[k] -= 32;
23.         }
24.     }
25.     if (memcmp(name, filename, 8) == 0) {
26.         file.cluster = ((uint32_t)entry->starthi << 16) | ent
ry->startlow;
27.         file.dir.index = entry_index;
28.         file.dir.cluster = 2;
29.         return file;
30.     }
31.     ++entry;
32. }
33. printk("file not found\n");
34. return file;
35. }

```

## 5. 读取与写入文件

对于 FAT32 文件系统的文件读取，会调用到 `fat32_read` 函数，需要根据 `file->fat32_file` 中的信息（即 `open` 的时候获取的信息）找到文件内容所在的簇，然后读取出文件内容到 `buf` 中：

```

1.  uint32_t get_file_size(struct file* file) {
2.      uint64_t sector = cluster_to_sector(file->fat32_file.dir.clus
ter) + file->fat32_file.dir.index / FAT32_ENTRY_PER_SECTOR;
3.      virtio_blk_read_sector(sector, fat32_table_buf);
4.      uint32_t index = file->fat32_file.dir.index % FAT32_ENTRY_PER
_SECTOR;
5.      return ((struct fat32_dir_entry *)fat32_table_buf)[index].siz
e;
6.  }
7.
8.  uint64_t fat32_read(struct file* file, void* buf, uint64_t len) {
9.      uint32_t file_size = get_file_size(file);
10.     uint64_t read_len = 0;
11.     while (read_len < len && file->cfo < file_size) {
12.         uint32_t cluster = file->fat32_file.cluster;
13.         for (uint32_t clusteri = 0; clusteri < file->cfo / (fat32
_volume.sec_per_cluster * VIRTIO_BLK_SECTOR_SIZE) && cluster < 0xFFF
FFFF8; ++clusteri) {
14.             //cluster += 1;
15.             cluster = next_cluster(cluster);
16.         }
17.         uint64_t sector = cluster_to_sector(cluster);

```

```

18.         uint64_t offset_in_sector = file->cfo % VIRTIO_BLK_SECTOR
    _SIZE;
19.         uint64_t remain_readable_size = VIRTIO_BLK_SECTOR_SIZE -
    offset_in_sector;
20.         if (remain_readable_size > len - read_len)
21.             remain_readable_size = len - read_len;
22.         if (remain_readable_size > file_size - file->cfo)
23.             remain_readable_size = file_size - file->cfo;
24.         virtio_blk_read_sector(sector, fat32_buf);
25.         memset(buf, 0, len - read_len);
26.         memcpy(buf, fat32_buf + offset_in_sector, remain_readable
    _size);
27.
28.         file->cfo += remain_readable_size;
29.         buf = (char *)buf + remain_readable_size;
30.         read_len += remain_readable_size;
31.
32.     }
33.     return read_len;
34.     /* todo: read content to buf, and return read length */
35. }
36.
37. int64_t fat32_write(struct file* file, const void* buf, uint64_t
    len) {
38.     uint64_t write_len = 0;
39.     while (len > 0) {
40.         uint32_t cluster = file->fat32_file.cluster;
41.         for (uint32_t clusteri = 0; clusteri < file->cfo / (fat32
    _volume.sec_per_cluster * VIRTIO_BLK_SECTOR_SIZE) && cluster < 0xFF
    FFFF8; ++clusteri) {
42.             cluster = next_cluster(cluster);
43.         }
44.         uint64_t sector = cluster_to_sector(cluster);
45.         uint64_t offset_in_sector = file->cfo % VIRTIO_BLK_SECTOR
    _SIZE;
46.         uint64_t remain_writable_size = VIRTIO_BLK_SECTOR_SIZE -
    offset_in_sector;
47.         if (remain_writable_size > len) {
48.             remain_writable_size = len;
49.         }
50.         virtio_blk_read_sector(sector, fat32_buf);
51.         memcpy(fat32_buf + offset_in_sector, buf, remain_writable
    _size);
52.         virtio_blk_write_sector(sector, fat32_buf);

```

```

53.
54.     file->cfo += remain_writable_size;
55.     buf += remain_writable_size;
56.     len -= remain_writable_size;
57.     write_len += remain_writable_size;
58. }
59.     return write_len;
60.     /* todo: fat32_write */
61. }

```

## 6. lseek 操作

在 nish 处理 edit 的时候，会先进行 lseek 调整文件指针，然后再进行 write。而 lseek 做的就是调整指针 file->cfo 的值：

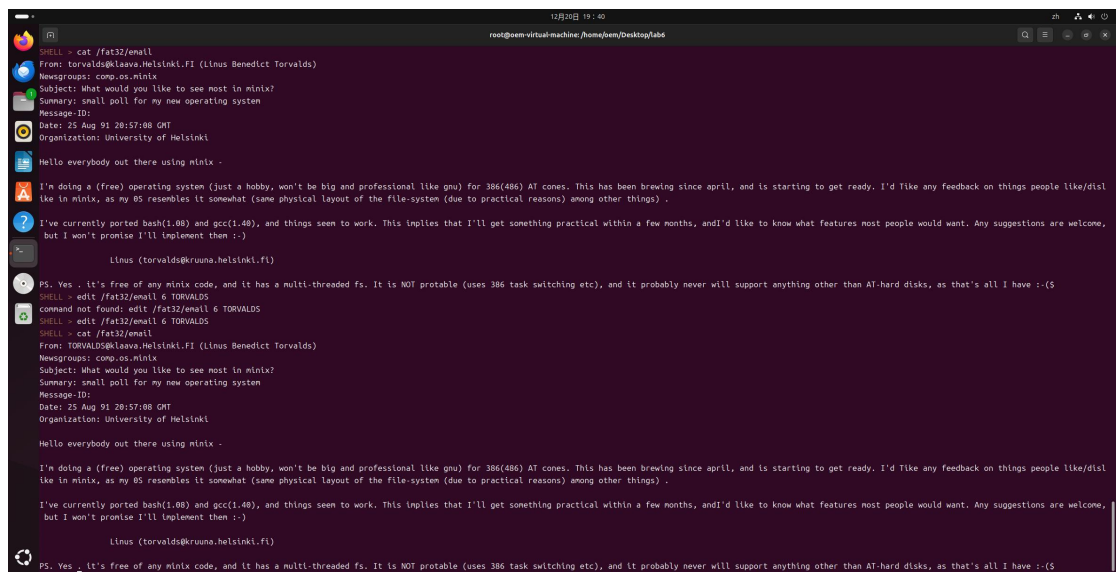
```

1.     int64_t fat32_lseek(struct file* file, int64_t offset, uint64_t whence) {
2.         if (whence == SEEK_SET) {
3.             file->cfo = offset/* to calculate */;
4.         } else if (whence == SEEK_CUR) {
5.             file->cfo = file->cfo + offset/* to calculate */;
6.         } else if (whence == SEEK_END) {
7.             /* Calculate file length */
8.             file->cfo = offset + get_file_size(file);/* to calculate */;
9.         } else {
10.            printk("fat32_lseek: whence not implemented\n");
11.            while (1);
12.        }
13.        return file->cfo;
14.    }

```

## （四）测试

测试结果如下：



```
root@osm-virtual-machine: /home/osm/Desktop/lab5
SHELL -> cat /fat32/email
From: torvalds@kruuna.helsinki.fi (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:00 GMT
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things) .

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)
```

```
Linus (torvalds@kruuna.helsinki.fi)
PS: Yes . it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-hard disks, as that's all I have :-)
```

```
SHELL -> edit /fat32/email 6 TORVALDS
command not found: edit /fat32/email 6 TORVALDS
SHELL -> edit /fat32/email 6 TORVALDS
SHELL -> cat /fat32/email
From: TORVALDS@kruuna.helsinki.fi (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:00 GMT
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things) .

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)
```

```
Linus (torvalds@kruuna.helsinki.fi)
PS: Yes . it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-hard disks, as that's all I have :-)
```

根据图中显示，能够正确实现读取文件和修改文件的功能，测试结果正常。

### 三、讨论和心得

本次实验实现了文件系统的操作，需要重点理解文件数据结构各个变量的意义。在本次实验中，有以下几个需要注意的点，而这正是实验指导里没有的，首先在 lab5 基础上添加 fs 目录时，除了修改 risc/kernel 下的 Makefile，还要修改根目录下的 Makefile，不然会导致 fs 目录下的文件不编译，其次给出的代码中用到了 memcpy，memcmp 等函数，但是直接编译会出现函数未定义的 bug，同时只引入标准库<string.h>也是不行的，需要在 lib/string.c 中重新写一遍这些函数的代码，还有在实现 fat32\_read, fat32\_write 等函数后，一定要在 syscall.c 中添加对应的系统调用，最后修改 Makefile 命令启动 VirtIO 接口的代码仅仅复制过来是不行的，一定要将缩进的四个空格删掉改为用 TAB 输入。

这次实验大大加深了我对 FAT32 文件系统结构的理解，对簇、扇区等概念有了更多认识。

### 四、思考题

无

## 五、附录

无